**->https://blog.csdn.net/redseazhaojianertao/article/details/53843454**

**VxWorks内核解读-3**

2016年12月23日 20:11:33 [redseazhaojianertao](https://me.csdn.net/redseazhaojianertao" \t "_blank) 阅读数：1051

本篇文章分析Wind内核调度器的设计原理以及其工作流程，设计支持多任务RTOS的关键是设计调度器，Wind内核调度器的目标是保证优先级最高的就绪任务处于运行状态。为了达到这一目的，需要在Wind内核的调度点判断就绪队列中优先级最高的任务是否正在运行，如果不在运行，调度器就会让这个优先级最高的任务抢占正在运行任务的CPU。

保证就绪队列中优先级最高的任务始终占据CPU是Wind内核可抢占的实质，其采用位图查找算法来定位最高优先级的进程，该算法的时间复杂度为O(1)，且与系统当前任务总数无关，即与系统负载无关，但是与系统支持的优先级数有关。Wind内核的调度点分成两处，一处是从内核态返回(调用windExit())，另一处是中断返回(调用intExit())。

**3.1 Wind内核调度器工作原理**

Wind内核调度器的作用：

* 使得任务状态的改变完全依赖于事件和系统调用的发生；
* 将一个任务从一个队列移动到另一个队列；
* 如果存在的话，执行任务切换(switch)和交换(swap)钩子函数,其中任务交换钩子函数仅仅提供了wind内核使用，任务切换钩子函数是提供给用户使用；
* 当系统空闲的时自旋寻找内部工作队列的空闲工作(Job)来执行；
* 具有电源管理接口。

在Wind内核中调度器reschedule()的实现有两个版本，一个是用C实现新的可移植性版本，一个是具体平台汇编实现(比如x86平台汇编)的优化版本。为了便于分析，我们选择C实现的可移植版本

reschedule()调度目的是选择合适的任务到CPU上运行，然后调用切换(switch)和交换(swap)钩子函数，最后载入挑选出来的任务的上下文。比较复杂的因素在于内核工作队列(kernel work queue)在reschedule()调用结束之前必须被检查是否为空。在C的可移植性版本中，检查内核工作队列和加载任务的上下文是在关中断的情况下完成的，这样做的目的是避免中断ISR的竞争。而在reschedule()的优化版本中，将加载任务上下文的部分尽可能的放在关中断和检查工作队列是否为空之前，这将会极大的降低中断延迟时间。

如果没有任务就绪，Wind内核将会在idle状态中不断的检查内核工作队列中是否有工作Job要做。由于Wind内核的idle状态不在任务的上下文当中，所以切换和交换钩子函数在内核处于idle状态时均是不能被调用的。一旦离开idle状态，如果新挑选的任务不同于内核进入idle状态时正在执行的任务(即存在新的更高优先级的任务就绪)，那么切换和交换钩子函数都会被调用。

代码如下(为了方便阅读，我删除了不相干的代码)：

void reschedule (void)

{

    FAST WIND\_TCB     \*taskIdPrevious;

    FAST int                ix;

    FAST UINT16          mask;

    int                              oldLevel;

unlucky:

    taskIdPrevious = taskIdCurrent;               /\* 记住旧的任务 \*/

    workQDoWork ();                                        /\* 执行内核队列中的所有Job ,清空内核队列\*/

    /\*在此处标记idle状态，直到有任务准备执行\*/

    kernelIsIdle = TRUE;                /\* 标记此时处于idle状态 \*/

    while (((WIND\_TCB \*) Q\_FIRST (&readyQHead)) == NULL)

     {

             workQDoWork ();/\*就绪队列为空，空转寻找内核队列中的Job执行\*/

     }

/\*执行到此处，说明就绪队列不空，有新的任务就绪\*/

taskIdCurrent = (WIND\_TCB \*) Q\_FIRST (&readyQHead);/\*挑选优先级最高的任务\*/

kernelIsIdle = FALSE;                       /\* 标记此时脱离idle状态 \*/

    /\* taskIdCurrent 指向将要运行的任务，如果其和

     \* taskIdPrevious保存的旧任务不同，内核将会执行切换和交换钩子函数.

     \*/

    if (taskIdCurrent != taskIdPrevious)         /\*如果不同则意味着是一个新的任务 \*/

    {

         /\* 执行钩子函数\*/

         mask = taskIdCurrent->swapInMask | taskIdPrevious->swapOutMask;

         for (ix = 0; mask != 0; ix++, mask = mask << 1)

             if (mask & 0x8000)

                   (\* taskSwapTable[ix]) (taskIdPrevious, taskIdCurrent);

         /\* do switch hooks \*/

         for (ix = 0; (ix < VX\_MAX\_TASK\_SWITCH\_RTNS) && (taskSwitchTable[ix] != NULL)；++ix)

             {

                     (\* taskSwitchTable[ix]) (taskIdPrevious, taskIdCurrent);

             }

    }

    oldLevel = intLock ();                           /\* 关中断 \*/

    if (!workQIsEmpty)

    {                                                                /\*内核队列中又有新的Job\*/

         intUnlock (oldLevel);                           /\* 开中断\*/

         goto unlucky;                                        /\* 从顶部重新开始执行 \*/

    }

    else

    {

         kernelState = FALSE;                           /\* 退出内核态 \*/

         /\*现在我们恢复调度器reschedule()选中的任务上下文，从代码中我们可以看出：整个

         \*恢复任务上下文和判断内核队列为空的操作都是在关中断的情况下进行的，在具体

         \*平台的汇编优化版本中，将会把恢复任务上下文的过程放入到关中断的前面，这样

         \*仅有判断内核队列是否为空的操作处于关中断的情况下，这将极大减低中断时延！

         \*/

         windLoadContext ();                           /\* dispatch the selected task \*/

     }

}

备注：这里需要指出的是，为什么在reschedule()的末尾再次判断内核队列是否为空，如果非空重新执行调度器呢？

这是因为，在执行内核切换和交换钩子函数期间，很有可能会有更高优先级的任务需要被唤醒，由于此时Wind内核正在被reschedule()调度器互斥访问(即Wind内核处于内核态，kernelState=TRUE)，将这些任务转为就绪态的工作都会被赛道内核队列延迟运行。如果在reschedule()的末尾检查内核队列非空，则证实了这种判断，reschedule()需要重新执行内核队列中的Job，使的更高优先级的任务可以进入就绪态，因而可以被提供一次调度运行的机会，这样确保了只有优先级最高的任务占据CPU。

另外，从Wind内核调度器的实现过程可以看出正在CPU上运行的任务仍然在就绪队列中，Wind内核这样处理有一个优点，如果正在运行的任务是优先级最高的任务，其会占据优先级队列的队首；如果其已经不是优先级最高的任务，将不会在优先级队列的首部。这样Wind内核只需要将优先级队列的队首元素和正在运行的任务比较，如果两者不同则说明当前任务已经不是优先级最高的任务，队首任务此时是优先级最高的任务，需要进行任务切换！

Wind内核调度器工作流程如图3.1。

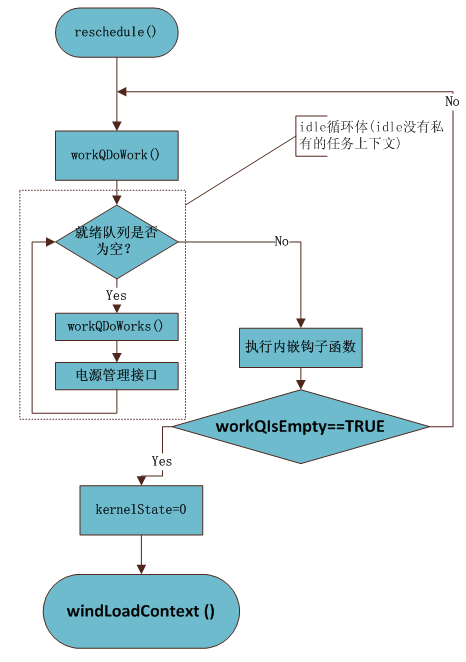


图3.1 Wind内核调度器工作流程

**3.2 调度器调度时机**

调度器reschedule()在Wind内核中仅被两处例程调用：退出内核的函数例程windExit()和退出中断的函数例程intExit()。这里需要指出的是Wind内核用一个全局变量kernelState来标志Wind内核是否进入内核态。

进入内核态，只需要简单地置kernelState为TRUE；

退出内核，顾名思义只需要将kernelState置为FALSE即可，但是退出内核态涉及到内核队列的清空操作，从而引发新的任务就绪，wind内核为了从形式上统一这个过程，将其封装在windExit()函数例程中，由于reschedule()调度器也是被windExit()调用，因此我们可以从更高的角度来说，退出内核的例程统一为windExit()例程

为了清楚地描述Wind内核的调度器reschedule()在wind内核的地位，我以时钟中断为例，来考察Wind内核的运行机制。我们首先描述一下时钟中断的流程，当Wind内核运行的某一个时刻，时钟中断发生时(暂不考虑时钟中断嵌套的情况），执行流程如下：

* 步骤1: Wind内核首先调用intEnt()例程保存当前任务的上下文到中断栈中，这一步是在关中断的情况下执行；
* 步骤2:Wind内核执行时钟中断处理函数tickAnnounce()(这里我去除了一系列的封装，VxWorks时钟中断实际的调用过程是sysClkInt()->usrClock()->tickAnnounce())：

void tickAnnounce (void)

{

    if (kernelState)               /\* defer work if in kernel \*/

                  workQAdd0 ((FUNCPTR)windTickAnnounce);

    else

         {

                  kernelState = TRUE;      /\* KERNEL\_ENT \*/

                  windTickAnnounce ();     /\* advance tick queue \*/

                  windExit ();                          /\* KERNEL\_EXIT \*/

         }

}

这一步是在开中断的条件下执行的；

* 步骤3:执行退出中断处理函数intExit()

**第一种情况：**

假设发生时钟中断的时刻，Wind内核当前不处于内核态(kernelStare=FALSE)时，则时钟中断处理函数通过将kernelState置为TRUE进入内核，直接执行wind内核态时钟中断处理例程windTickAnnounce()，执行完毕后调用windExit()退出内核态。windExit的执行流程如图3.2。

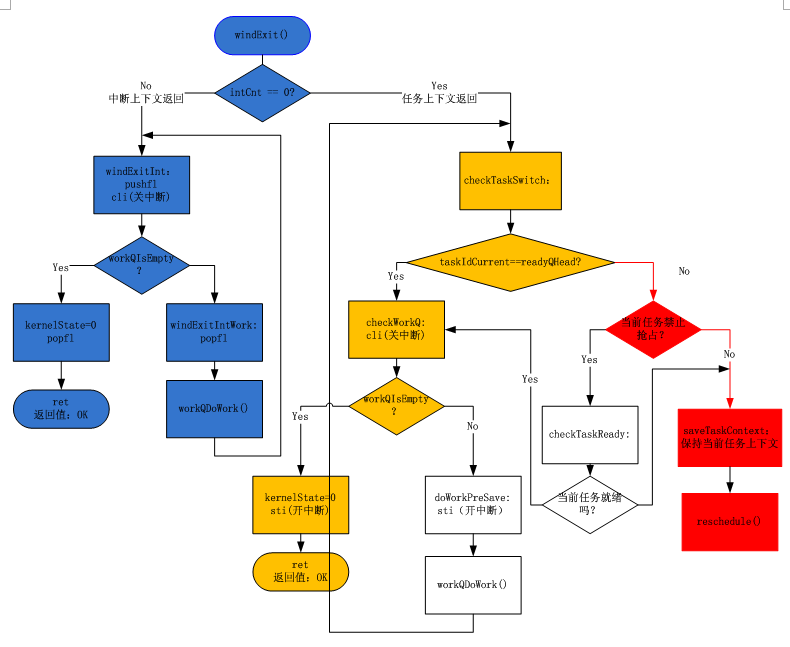


图3.2 windExit()执行流程图

由于此时windExit()还在时钟中断上下文中，执行图中左边的流程:

1. 如果内核队列为空，则开中断、退出内核态，跳转到3；
2. 如果内核队列非空，先处理内核队列中的Job，处理完毕后跳转到2；
3. 返回；

接着执行intExit()退出中断处理函数，intExit()处理流程如图3.3。

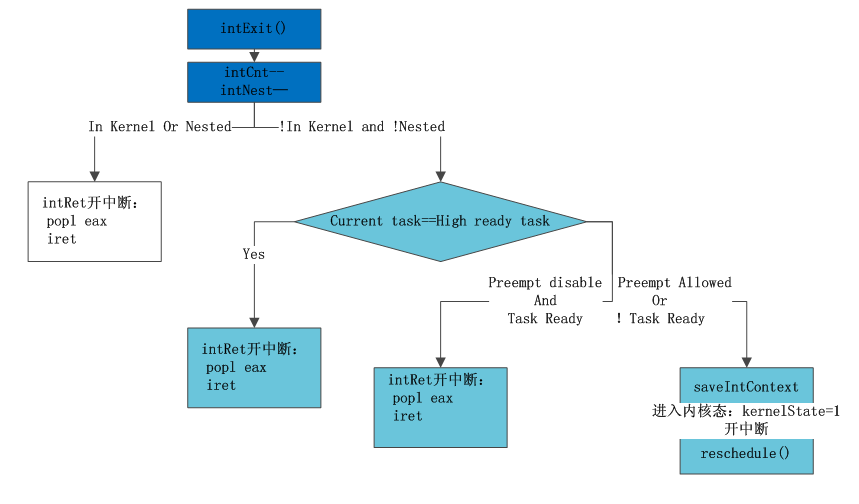


图3.3 intExit()退出中断处理流程

此时如果被中断任务仍然是优先级最高的任务，则恢复被中断任务的上下文，继续执行；

如果存在更高优先级的任务就绪，但被中断的任务禁止抢占，且处于就绪态，则恢复被中断任务的上下文，继续执行；如果被中断的任务支持抢占或者主动放弃CPU，则恢复最高优先级的任务。

如果时钟中断发生嵌套时，发生的阶段只能在正在执行核心中断处理例程期间(tickAnnounce()执行期间)，因为这一阶段是开中断，并且此阶段正处于Wind内核的内核态中。根据tickAnnounce()的处理流程，其会将Wind内核中断服务例程作为一个Job直接放置到内核工作队列中返回，接着执行intExit()函数。由于此时正处于中断嵌套阶段，根据intExit()处理流程，其会进一步返回到上一层中断中继续执行。

那么本次中断的处理程序windTickAnnounce() 什么时候进行处理呢？

我们知道最外层中断处理函数占据着wind的内核态，当其调用windExit()从内核态退出时,根据windExit()的处理流程，其会判断内核队列是否为空，在内核队列不空的情况下，执行内核队列中的Job，清空内核队列,此时嵌套的中断得到处理。

备注：从这里的处理流程我们可以看出wind内核中断嵌套处理程序均是采用延后处理的机制，处理的时机在于最外层的中断退出内核态之前(windExit()执行流程)。

**第二种情况：**

中断发生的时刻，Wind内核正处于内核态。正在内核态访问的代码可以是一段中断ISR，我们在上面已经分析过；也可以是一个任务正在调用内核态例程，比如正在执行idle空转。这种情况只能发生在当前任务从就绪态转换为阻塞态，并且此时就绪队列为空的情况下。

**第三种情况：**

中断发生的时刻，Wind内核正处于非内核态，此时处于最高优先级的任务正在执行。此时时钟中断的执行流程是：

* 第一步：执行intEnt()例程，该函数例程关中断，保存当前任务的上下文到中断栈中；
* 第二步：执行中断处理函数：

VxWorks时钟中断实际的调用过程是sysClkInt()->usrClock()->tickAnnounce())：

void tickAnnounce (void)

{

    if (kernelState)                            /\* defer work if in kernel \*/

                  workQAdd0 ((FUNCPTR)windTickAnnounce);

    else

         {

                  kernelState = TRUE;                  /\* KERNEL\_ENT \*/

                  windTickAnnounce ();                /\* advance tick queue \*/

                  windExit ();                          /\* KERNEL\_EXIT \*/

         }

}

这一步是在开中断的条件下执行的，由于此时内核态没有被占用(kernelState=FALSE)，不需要放在内核延迟队列中，直接执行windTickAnnounce()

* 第三步：中断返回intExit()，如图3.3所示。

由于此时中断没有嵌套，intExit()将执行右边的流程。

此时如果被中断任务仍然是优先级最高的任务，则恢复被中断任务的上下文继续执行；如果期间由于中断ISR的执行，导致了新的更高优先级的任务就绪的话，intExit()将会根据情况走两个分支：

* 第一个：如果被中断任务禁止抢占，并处于就绪态，则恢复被中断任务继续执行；
* 第二个：如果被中断任务允许抢占，或者被中断任务已经处于非就绪态，则执行调度器挑选优先级最高的任务执行。

这里大家可以存在一个疑问，由于这个场景中中断ISR直接进入内核态执行中断处理函数windTickAnnounce ()，执行完毕后就可能存在更高优先级的任务就绪，紧接着中断ISR执行退出内核态例程windExit()。由于windExit()存在一个调度点。更高优先级的任务会不在这里进行调度，而不需要等到中断返回intExit()的时刻？我们可以分析一下在这种情景下的windExit()执行流程，为了方便描述，再次画一下WindExit()的执行流程，如图3.4。

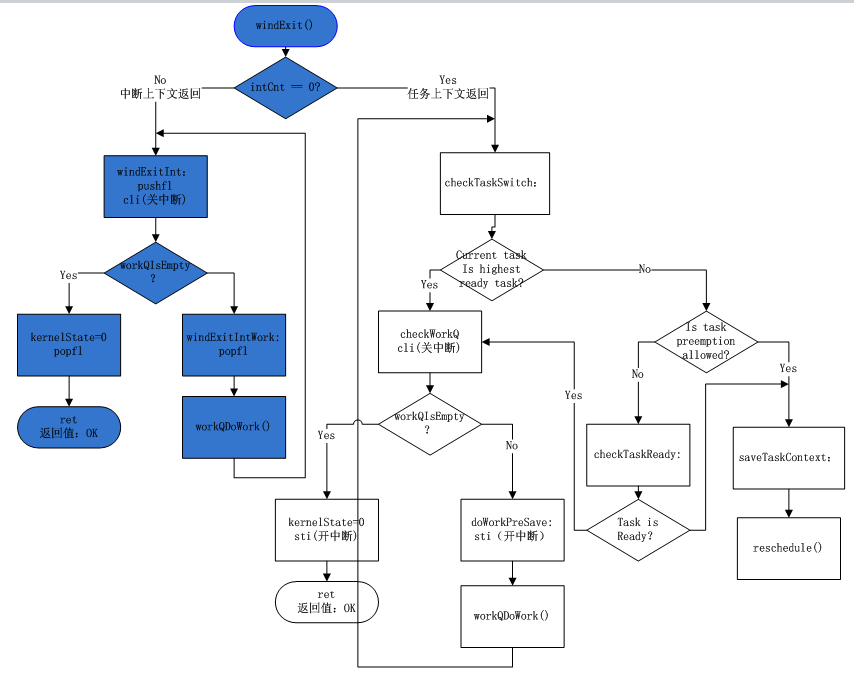


图3.4 WindExit()的执行流程

从图中我们可以看出，由于此时仍处在中断上下文中，windExit()走左边的分支，这里只可能会做的就是执行内核Job来清空内核队列，调用完毕后直接返回。此时的windExit()充当一个函数例程的角色，不涉及到抢占点的引入。因而只有可能在intExit()中才会给新就绪的任务一个执行的机会。

**第四种情况：**

任务在执行的过程中，需要调用内核态例程的服务，比如延迟自己，创建一个新任务等等，具体的内核例程如下：

1.       void windSpawn(FAST WIND\_TCB \*pTcb):创建一个任务，并将该任务置为阻塞态(suspend state),放置到活动队列（active queue）中。

2.       STATUS windDelete(FAST WIND\_TCB \*pTcb):删除任务，并重新排列该任务所在的任何队列，该例程假设被删除的任务不具有任务遗传性的信号量。

3.       void windSuspend(FAST WIND\_TCB \*pTcb):阻塞(suspend)一个任务，阻塞态(suspension)是一个附加的状态。当一个任务处于就绪队列中时，其将会从中移除；该阻塞的任务不是处于就绪态时，其将会在已有的状态上再加上阻塞态（suspension）。

4.       void windResume(FAST WIND\_TCB \*pTcb):唤醒一个指定的任务，如果必要的话，将其放置到就绪队列中。

5.       oid windPriNormalSet(WIND\_TCB \*pTcb, UINT priNormal):设置任务的正常优先级数，如果不发生优先级继承，任务按照其正常优先级数执行。

6.       void windPrioritySet(FAST WIND\_TCB \*pTcb, FAST UINT priority):设置任务的实际优先级数，并且考虑到实际的优先级翻转安全性。如果任务拥有任何的优先级翻转安全信号量，那么优先级将不允许降低。

7.       void windSemDelete(FAST SEM\_ID semId):删除一个信号量，并将在该信号量上等待的所有任务解除阻塞。

8.       void windTickAnnounce(void):处理延迟队列，使得延迟时间到期的任务就绪。如果配置了时间片轮转策略，则执行时间片轮转调度策略；通知执行任务到期的看门狗程序。

9.       STATUS windDelay(FAST int timeout):将任务根据延迟的长度timeout插入到延时队列的合适位置。

10.   STATUS windUndelay(WIND\_TCB \*pTcb):将睡眠的任务唤醒

11.   STATUS windWdStart(WDOG \*wdId, int timeout):启动或者重启一个开门够。如果看门狗已经在定时队列（tickQHead） 中，它将根据timeout的值重启排序。如果在内核队列中存在多个windWdStart() Job，正如变量deferStartCn所计数的那样，windWdStart()函数将什么也不做，直接返回。

12.   void windWdCancel(WDOG \*wdId):取消看门狗，如果需要的话，将其从定时器队列中移除。

13.   void windPendQGet(Q\_HEAD \*pQHead):从pQHead指定的等待队列中取出队头任务，清除其等待状态，如果其处于延时状态，将其从延时状态清除，并剔除出延时队列。如果其原来处于就绪态，则将其放入就绪队列。

14.   void windReadyQPut(WIND\_TCB \*pTcb):将一个先前在某个共享信号量上阻塞的任务放到就绪队列中，其具体做法是将这个任务的TCB控制块从共享信号量的等待队列中移除（由释放该共享信号量的CPU负责处理），如果其处于延时状态，将其从延时状态清除，并剔除出延时队列。如果其原来处于就绪态，则将其放入就绪队列。

15.   void windReadyQRemove(Q\_HEAD \*pQHead, int timeout):将当前任务从就绪队列中移除，并将其置为等待态(WIND\_PEND)放到pQHead指向的等待队列中去；如果该任务被定时器，则也将其放入到定时队列的相应位置。

16.   void windPendQFlush(Q\_HEAD \*pQHead):将在pQHead指向的等待队列中的所有任务结束等待状态，如果其本来就处于就绪态，则将其放置到就绪队列当中。

17.   STATUS windPendQPut(FAST Q\_HEAD \*pQHead, FAST int timeout):将当前任务放置到pQHead指向的等待队列当中，如果当前任务被定时，则同时放置到延时队列当中。如果timeout的值是NO\_WAIT则返回ERROR。

18.   STATUS windPendQRemove(WIND\_TCB \*pTcb):将pTcb指定的任务从等待队列中移除。

19.   void windPendQTerminate(Q\_HEAD \*pQHead):将pQHead指定的等待队列中的所有任务移除等待队列。这些任务清除其等待位(WIND\_PEND),如果是被延时，则将其从延时队列中移除，如果其原来处于就绪态，则将其重新放置到就绪队列。

备注：从STATUS windPendQPut(FAST Q\_HEAD \*pQHead, FAST int timeout)的实现中，我们可以看出，如果当前任务因为等待某一个信号量而被放置到该信号量的等待队列中去时，只是设置该任务的WIND\_PEND位，如果其再被要求等待一段时间timeout的话，也只设置该任务的WIND\_DELAY位，并放置到延时队列中去。特别需要指出的是其WIND\_READY位并没有被清除，这意味着，当将一个任务从等待队列中移除时，只需要查看该任务的WIND\_READY位，就可以确定该任务在放入到等待队列之前是否处于就绪状态，如果处于就绪状态就放置到就绪队列中。windLib库中操作等待队列的例程，均利用到了这一技巧，windPendQPut()我这里就不粘贴了，节省篇幅。

**3.3 wind内核态分析**

从3.2节我们可以看出Wind内核的内核态例程一共包含了19个例程，这些例程构成了Wind内核最基本的服务，由全局变量kernelState包含。只有在kernelState变量为TRUE是，外围例程才能调用这些服务例程。冲这个角度来说，虽然Wind内核运行在处理器的特权态。当时kernelState利用模拟了一个更高的特权态。该特权态外围软件只能互斥访问，并且禁止抢占。

大家可能有一个疑问，这些内核态例程禁止抢占，这会不会影响到这个VxWorks系统的响应时间呢？答案是肯定会的，影响的程度在于这些内核例程的设计，这19个内核例程的设计非常的精简，仅仅完成最核心的功能。并且当Wind内核处于内核态时，尽管禁止强占，但是中断是开着的，仍然可以响应中断。

Wind内核的创新之处在于设计了有限长度的环形内核队列(Wind2.6版本设置为为64)。当Wind内核在内核态时发作中断时，中断的上下文的保存和恢复仍然正常处理，但是把真正的中断处理函数作为一个Job放置到内核队列中，等到Wind内核退出内核态时在做处理，设置内核队列的目的也就在此，目的是延后处理中断ISR。

这从我们对时钟中断的分析可以窥见全貌，大家或许还有一个疑问那就是？这种延时处理中断ISR的机制会不会影响到VxWorks系统的中断响应时间呢？

答案是肯定的，毕竟中断ISR只有等到其退出内核态才得到处理嘛，但是问题的关键在于Wind内核处于内核态的这19个例程非常精简，其在内核态的时间非常短。这种处理机制和Linux内核中将中断分成上下半段,下半段（称之为软中断）延后处理的机制道理是相同的。

这同时也告诉我们,在设置VxWorks的具体外设的中断处理框架时，必须分清楚哪些是必须处理的(比如最基本的是写EOI命令），哪些是可以放在内核队列中延后执行。这些是提高VxWorks实时性能的关键。

Wind内核进入内核态只需要将kernelState置为TRUE即可，退出内核态时则需要调用windExit()例程，这两个例程需要配套使用。

此次，关于Wind内核的调度模块就分析到这来了，大家如果有疑问或者不清楚的地方，可以给我留言，O(∩\_∩)O~

待续。。。。。